carousel _controller.dart –

```dart
import 'dart:async';

import 'package:flutter/material.dart';

import 'carousel_options.dart';
import 'carousel_state.dart';
import 'utils.dart';

abstract class CarouselController {
 bool get ready;

 Future<Null> get onReady;

 Future<void> nextPage({Duration? duration, Curve? curve});

 Future<void> previousPage({Duration? duration, Curve? curve});

 void jumpToPage(int page);

 Future<void> animateToPage(int page, {Duration? duration, Curve? curve});

 void startAutoPlay();

 void stopAutoPlay();

 factory CarouselController() => CarouselControllerImpl();
}

class CarouselControllerImpl implements CarouselController {
 final Completer<Null> _readyCompleter = Completer<Null>();

 CarouselState? _state;

 set state(CarouselState? state) {
   _state = state;
   if (!_readyCompleter.isCompleted) {
     _readyCompleter.complete();
   }
 }

 void _setModeController() =>
     _state!.changeMode(CarouselPageChangedReason.controller);

 @override
 bool get ready => _state != null;

 @override
 Future<Null> get onReady => _readyCompleter.future;

 /// Animates the controlled [CarouselSlider] to the next page.
 ///
```

```dart
 /// The animation lasts for the given duration and follows the given
curve.
 /// The returned [Future] resolves when the animation completes.
 Future<void> nextPage(
     {Duration? duration = const Duration(milliseconds: 300),
     Curve? curve = Curves.linear}) async {
   final bool isNeedResetTimer =
_state!.options.pauseAutoPlayOnManualNavigate;
   if (isNeedResetTimer) {
     _state!.onResetTimer();
   }
   _setModeController();
   await _state!.pageController!.nextPage(duration: duration!, curve:
curve!);
   if (isNeedResetTimer) {
     _state!.onResumeTimer();
   }
 }

 /// Animates the controlled [CarouselSlider] to the previous page.
 ///
 /// The animation lasts for the given duration and follows the given
curve.
 /// The returned [Future] resolves when the animation completes.
 Future<void> previousPage(
     {Duration? duration = const Duration(milliseconds: 300),
     Curve? curve = Curves.linear}) async {
   final bool isNeedResetTimer =
_state!.options.pauseAutoPlayOnManualNavigate;
   if (isNeedResetTimer) {
     _state!.onResetTimer();
   }
   _setModeController();
   await _state!.pageController!
       .previousPage(duration: duration!, curve: curve!);
   if (isNeedResetTimer) {
     _state!.onResumeTimer();
   }
 }

 /// Changes which page is displayed in the controlled [CarouselSlider].
 ///
 /// Jumps the page position from its current value to the given value,
 /// without animation, and without checking if the new value is in range.
 void jumpToPage(int page) {
   final index = getRealIndex(_state!.pageController!.page!.toInt(),
       _state!.realPage - _state!.initialPage, _state!.itemCount);

   _setModeController();
   final int pageToJump = _state!.pageController!.page!.toInt() + page -
index;
   return _state!.pageController!.jumpToPage(pageToJump);
 }
```

```dart
 /// Animates the controlled [CarouselSlider] from the current page to the
given page.
 ///
 /// The animation lasts for the given duration and follows the given
curve.
 /// The returned [Future] resolves when the animation completes.
 Future<void> animateToPage(int page,
     {Duration? duration = const Duration(milliseconds: 300),
     Curve? curve = Curves.linear}) async {
   final bool isNeedResetTimer =
_state!.options.pauseAutoPlayOnManualNavigate;
   if (isNeedResetTimer) {
     _state!.onResetTimer();
   }
   final index = getRealIndex(_state!.pageController!.page!.toInt(),
       _state!.realPage - _state!.initialPage, _state!.itemCount);
   int smallestMovement = page - index;
   if (_state!.options.enableInfiniteScroll &&
       _state!.itemCount != null &&
       _state!.options.animateToClosest) {
     if ((page - index).abs() > (page + _state!.itemCount! - index).abs()) {
       smallestMovement = page + _state!.itemCount! - index;
     } else if ((page - index).abs() >
         (page - _state!.itemCount! - index).abs()) {
       smallestMovement = page - _state!.itemCount! - index;
     }
   }
   _setModeController();
   await _state!.pageController!.animateToPage(
       _state!.pageController!.page!.toInt() + smallestMovement,
       duration: duration!,
       curve: curve!);
   if (isNeedResetTimer) {
     _state!.onResumeTimer();
   }
 }

 /// Starts the controlled [CarouselSlider] autoplay.
 ///
 /// The carousel will only autoPlay if the [autoPlay] parameter
 /// in [CarouselOptions] is true.
 void startAutoPlay() {
   _state!.onResumeTimer();
 }

 /// Stops the controlled [CarouselSlider] from autoplaying.
 ///
 /// This is a more on-demand way of doing this. Use the [autoPlay]
 /// parameter in [CarouselOptions] to specify the autoPlay behaviour of
the carousel.
 void stopAutoPlay() {
   _state!.onResetTimer();
```

```
 }
}
```

Style.dart –

```
import 'dart:ui';

import 'package:flutter/material.dart';
import 'package:flutter_html/flutter_html.dart';
import 'package:flutter_html/src/css_parser.dart';

///This class represents all the available CSS attributes
///for this package.
class Style {
 /// CSS attribute "`background-color`"
 ///
 /// Inherited: no,
 /// Default: Colors.transparent,
 Color? backgroundColor;

 /// CSS attribute "`color`"
 ///
 /// Inherited: yes,
 /// Default: unspecified,
 Color? color;

 /// CSS attribute "`direction`"
 ///
 /// Inherited: yes,
 /// Default: TextDirection.ltr,
 TextDirection? direction;

 /// CSS attribute "`display`"
 ///
 /// Inherited: no,
 /// Default: unspecified,
 Display? display;

 /// CSS attribute "`font-family`"
 ///
 /// Inherited: yes,
 /// Default: Theme.of(context).style.textTheme.body1.fontFamily
 String? fontFamily;

 /// CSS attribute "`font-feature-settings`"
 ///
 /// Inherited: yes,
 /// Default: normal
 List<FontFeature>? fontFeatureSettings;

 /// CSS attribute "`font-size`"
 ///
 /// Inherited: yes,
```

```
/// Default: FontSize.medium
FontSize? fontSize;

/// CSS attribute "`font-style`"
///
/// Inherited: yes,
/// Default: FontStyle.normal,
FontStyle? fontStyle;

/// CSS attribute "`font-weight`"
///
/// Inherited: yes,
/// Default: FontWeight.normal,
FontWeight? fontWeight;

/// CSS attribute "`height`"
///
/// Inherited: no,
/// Default: Unspecified (null),
double? height;

/// CSS attribute "`letter-spacing`"
///
/// Inherited: yes,
/// Default: normal (0),
double? letterSpacing;

/// CSS attribute "`list-style-type`"
///
/// Inherited: yes,
/// Default: ListStyleType.DISC
ListStyleType? listStyleType;

/// CSS attribute "`list-style-position`"
///
/// Inherited: yes,
/// Default: ListStylePosition.OUTSIDE
ListStylePosition? listStylePosition;

/// CSS attribute "`padding`"
///
/// Inherited: no,
/// Default: EdgeInsets.zero
EdgeInsets? padding;

/// CSS attribute "`margin`"
///
/// Inherited: no,
/// Default: EdgeInsets.zero
EdgeInsets? margin;

/// CSS attribute "`text-align`"
///
```

```
 /// Inherited: yes,
 /// Default: TextAlign.start,
 TextAlign? textAlign;

 /// CSS attribute "`text-decoration`"
 ///
 /// Inherited: no,
 /// Default: TextDecoration.none,
 TextDecoration? textDecoration;

 /// CSS attribute "`text-decoration-color`"
 ///
 /// Inherited: no,
 /// Default: Current color
 Color? textDecorationColor;

 /// CSS attribute "`text-decoration-style`"
 ///
 /// Inherited: no,
 /// Default: TextDecorationStyle.solid,
 TextDecorationStyle? textDecorationStyle;

 /// Loosely based on CSS attribute "`text-decoration-thickness`"
 ///
 /// Uses a percent modifier based on the font size.
 ///
 /// Inherited: no,
 /// Default: 1.0 (specified by font size)
 // TODO(Sub6Resources): Possibly base this more closely on the CSS
attribute.
 double? textDecorationThickness;

 /// CSS attribute "`text-shadow`"
 ///
 /// Inherited: yes,
 /// Default: none,
 List<Shadow>? textShadow;

 /// CSS attribute "`vertical-align`"
 ///
 /// Inherited: no,
 /// Default: VerticalAlign.BASELINE,
 VerticalAlign? verticalAlign;

 /// CSS attribute "`white-space`"
 ///
 /// Inherited: yes,
 /// Default: WhiteSpace.NORMAL,
 WhiteSpace? whiteSpace;

 /// CSS attribute "`width`"
 ///
 /// Inherited: no,
```

```
/// Default: unspecified (null)
double? width;

/// CSS attribute "`word-spacing`"
///
/// Inherited: yes,
/// Default: normal (0)
double? wordSpacing;

/// CSS attribute "`line-height`"
///
/// Supported values: double values
///
/// Unsupported values: normal, 80%, ..
///
/// Inherited: no,
/// Default: Unspecified (null),
LineHeight? lineHeight;

//TODO modify these to match CSS styles
String? before;
String? after;
Border? border;
Alignment? alignment;
Widget? markerContent;

/// MaxLine
///
///
///
///
int? maxLines;

/// TextOverflow
///
///
///
///
TextOverflow? textOverflow;

TextTransform? textTransform;

Style({
  this.backgroundColor = Colors.transparent,
  this.color,
  this.direction,
  this.display,
  this.fontFamily,
  this.fontFeatureSettings,
  this.fontSize,
  this.fontStyle,
  this.fontWeight,
  this.height,
```

```
    this.lineHeight,
    this.letterSpacing,
    this.listStyleType,
    this.listStylePosition,
    this.padding,
    this.margin,
    this.textAlign,
    this.textDecoration,
    this.textDecorationColor,
    this.textDecorationStyle,
    this.textDecorationThickness,
    this.textShadow,
    this.verticalAlign,
    this.whiteSpace,
    this.width,
    this.wordSpacing,
    this.before,
    this.after,
    this.border,
    this.alignment,
    this.markerContent,
    this.maxLines,
    this.textOverflow,
    this.textTransform = TextTransform.none,
  }) {
    if (this.alignment == null &&
        (display == Display.BLOCK || display == Display.LIST_ITEM)) {
      this.alignment = Alignment.centerLeft;
    }
  }

  static Map<String, Style> fromThemeData(ThemeData theme) => {
    'h1': Style.fromTextStyle(theme.textTheme.headlineLarge!),
    'h2': Style.fromTextStyle(theme.textTheme.headlineMedium!),
    'h3': Style.fromTextStyle(theme.textTheme.headlineSmall!),
    'h4': Style.fromTextStyle(theme.textTheme.titleLarge!),
    'h5': Style.fromTextStyle(theme.textTheme.titleMedium!),
    'h6': Style.fromTextStyle(theme.textTheme.titleSmall!),
    'body': Style.fromTextStyle(theme.textTheme.bodyMedium!),
  };

  static Map<String, Style> fromCss(String css, OnCssParseError?
onCssParseError) {
    final declarations = parseExternalCss(css, onCssParseError);
    Map<String, Style> styleMap = {};
    declarations.forEach((key, value) {
      styleMap[key] = declarationsToStyle(value);
    });
    return styleMap;
  }

  TextStyle generateTextStyle() {
    return TextStyle(
```

```
    backgroundColor: backgroundColor,
    color: color,
    decoration: textDecoration,
    decorationColor: textDecorationColor,
    decorationStyle: textDecorationStyle,
    decorationThickness: textDecorationThickness,
    fontFamily: fontFamily,
    fontFeatures: fontFeatureSettings,
    fontSize: fontSize?.size,
    fontStyle: fontStyle,
    fontWeight: fontWeight,
    letterSpacing: letterSpacing,
    shadows: textShadow,
    wordSpacing: wordSpacing,
    height: lineHeight?.size ?? 1.0,
    //TODO background
    //TODO textBaseline
  );
}

@override
String toString() {
  return "Style";
}

Style merge(Style other) {
  return copyWith(
    backgroundColor: other.backgroundColor,
    color: other.color,
    direction: other.direction,
    display: other.display,
    fontFamily: other.fontFamily,
    fontFeatureSettings: other.fontFeatureSettings,
    fontSize: other.fontSize,
    fontStyle: other.fontStyle,
    fontWeight: other.fontWeight,
    height: other.height,
    lineHeight: other.lineHeight,
    letterSpacing: other.letterSpacing,
    listStyleType: other.listStyleType,
    listStylePosition: other.listStylePosition,
    padding: other.padding,
    //TODO merge EdgeInsets
    margin: other.margin,
    //TODO merge EdgeInsets
    textAlign: other.textAlign,
    textDecoration: other.textDecoration,
    textDecorationColor: other.textDecorationColor,
    textDecorationStyle: other.textDecorationStyle,
    textDecorationThickness: other.textDecorationThickness,
    textShadow: other.textShadow,
    verticalAlign: other.verticalAlign,
    whiteSpace: other.whiteSpace,
```

```
    width: other.width,
    wordSpacing: other.wordSpacing,

    before: other.before,
    after: other.after,
    border: other.border,
    //TODO merge border
    alignment: other.alignment,
    markerContent: other.markerContent,
    maxLines: other.maxLines,
    textOverflow: other.textOverflow,
    textTransform: other.textTransform,
  );
}

Style copyOnlyInherited(Style child) {
  FontSize? finalFontSize = child.fontSize != null ?
    fontSize != null && child.fontSize?.units == "em" ?
      FontSize(child.fontSize!.size! * fontSize!.size!) : child.fontSize
    : fontSize != null && fontSize!.size! < 0 ?
      FontSize.percent(100) : fontSize;
  LineHeight? finalLineHeight = child.lineHeight != null ?
    child.lineHeight?.units == "length" ?
      LineHeight(child.lineHeight!.size! / (finalFontSize == null ? 14 :
finalFontSize.size!) * 1.2) : child.lineHeight
    : lineHeight;
  return child.copyWith(
    backgroundColor: child.backgroundColor != Colors.transparent ?
      child.backgroundColor : backgroundColor,
    color: child.color ?? color,
    direction: child.direction ?? direction,
    display: display == Display.NONE ? display : child.display,
    fontFamily: child.fontFamily ?? fontFamily,
    fontFeatureSettings: child.fontFeatureSettings ?? fontFeatureSettings,
    fontSize: finalFontSize,
    fontStyle: child.fontStyle ?? fontStyle,
    fontWeight: child.fontWeight ?? fontWeight,
    lineHeight: finalLineHeight,
    letterSpacing: child.letterSpacing ?? letterSpacing,
    listStyleType: child.listStyleType ?? listStyleType,
    listStylePosition: child.listStylePosition ?? listStylePosition,
    textAlign: child.textAlign ?? textAlign,
    textDecoration: TextDecoration.combine(
        [child.textDecoration ?? TextDecoration.none,
          textDecoration ?? TextDecoration.none]),
    textShadow: child.textShadow ?? textShadow,
    whiteSpace: child.whiteSpace ?? whiteSpace,
    wordSpacing: child.wordSpacing ?? wordSpacing,
    maxLines: child.maxLines ?? maxLines,
    textOverflow: child.textOverflow ?? textOverflow,
    textTransform: child.textTransform ?? textTransform,
  );
}
```

```
Style copyWith({
  Color? backgroundColor,
  Color? color,
  TextDirection? direction,
  Display? display,
  String? fontFamily,
  List<FontFeature>? fontFeatureSettings,
  FontSize? fontSize,
  FontStyle? fontStyle,
  FontWeight? fontWeight,
  double? height,
  LineHeight? lineHeight,
  double? letterSpacing,
  ListStyleType? listStyleType,
  ListStylePosition? listStylePosition,
  EdgeInsets? padding,
  EdgeInsets? margin,
  TextAlign? textAlign,
  TextDecoration? textDecoration,
  Color? textDecorationColor,
  TextDecorationStyle? textDecorationStyle,
  double? textDecorationThickness,
  List<Shadow>? textShadow,
  VerticalAlign? verticalAlign,
  WhiteSpace? whiteSpace,
  double? width,
  double? wordSpacing,
  String? before,
  String? after,
  Border? border,
  Alignment? alignment,
  Widget? markerContent,
  int? maxLines,
  TextOverflow? textOverflow,
  TextTransform? textTransform,
  bool? beforeAfterNull,
}) {
  return Style(
    backgroundColor: backgroundColor ?? this.backgroundColor,
    color: color ?? this.color,
    direction: direction ?? this.direction,
    display: display ?? this.display,
    fontFamily: fontFamily ?? this.fontFamily,
    fontFeatureSettings: fontFeatureSettings ?? this.fontFeatureSettings,
    fontSize: fontSize ?? this.fontSize,
    fontStyle: fontStyle ?? this.fontStyle,
    fontWeight: fontWeight ?? this.fontWeight,
    height: height ?? this.height,
    lineHeight: lineHeight ?? this.lineHeight,
    letterSpacing: letterSpacing ?? this.letterSpacing,
    listStyleType: listStyleType ?? this.listStyleType,
    listStylePosition: listStylePosition ?? this.listStylePosition,
```

```
    padding: padding ?? this.padding,
    margin: margin ?? this.margin,
    textAlign: textAlign ?? this.textAlign,
    textDecoration: textDecoration ?? this.textDecoration,
    textDecorationColor: textDecorationColor ?? this.textDecorationColor,
    textDecorationStyle: textDecorationStyle ?? this.textDecorationStyle,
    textDecorationThickness:
        textDecorationThickness ?? this.textDecorationThickness,
    textShadow: textShadow ?? this.textShadow,
    verticalAlign: verticalAlign ?? this.verticalAlign,
    whiteSpace: whiteSpace ?? this.whiteSpace,
    width: width ?? this.width,
    wordSpacing: wordSpacing ?? this.wordSpacing,
    before: beforeAfterNull == true ? null : before ?? this.before,
    after: beforeAfterNull == true ? null : after ?? this.after,
    border: border ?? this.border,
    alignment: alignment ?? this.alignment,
    markerContent: markerContent ?? this.markerContent,
    maxLines: maxLines ?? this.maxLines,
    textOverflow: textOverflow ?? this.textOverflow,
    textTransform: textTransform ?? this.textTransform,
  );
 }

 Style.fromTextStyle(TextStyle textStyle) {
   this.backgroundColor = textStyle.backgroundColor;
   this.color = textStyle.color;
   this.textDecoration = textStyle.decoration;
   this.textDecorationColor = textStyle.decorationColor;
   this.textDecorationStyle = textStyle.decorationStyle;
   this.textDecorationThickness = textStyle.decorationThickness;
   this.fontFamily = textStyle.fontFamily;
   this.fontFeatureSettings = textStyle.fontFeatures;
   this.fontSize = FontSize(textStyle.fontSize);
   this.fontStyle = textStyle.fontStyle;
   this.fontWeight = textStyle.fontWeight;
   this.letterSpacing = textStyle.letterSpacing;
   this.textShadow = textStyle.shadows;
   this.wordSpacing = textStyle.wordSpacing;
   this.lineHeight = LineHeight(textStyle.height ?? 1.2);
   this.textTransform = TextTransform.none;
 }
}

enum Display {
 BLOCK,
 INLINE,
 INLINE_BLOCK,
 LIST_ITEM,
 NONE,
}

class FontSize {
```

```
 final double? size;
 final String units;

 const FontSize(this.size, {this.units = ""});

 /// A percentage of the parent style's font size.
 factory FontSize.percent(int percent) {
   return FontSize(percent.toDouble() / -100.0, units: "%");
 }

 factory FontSize.em(double? em) {
   return FontSize(em, units: "em");
 }

 factory FontSize.rem(double rem) {
   return FontSize(rem * 16 - 2, units: "rem");
 }
 // These values are calculated based off of the default (`medium`)
 // being 14px.
 //
 // TODO(Sub6Resources): This seems to override Flutter's accessibility text
scaling.
 //
 // Negative values are computed during parsing to be a percentage of
 // the parent style's font size.
 static const xxSmall = FontSize(7.875);
 static const xSmall = FontSize(8.75);
 static const small = FontSize(11.375);
 static const medium = FontSize(14.0);
 static const large = FontSize(15.75);
 static const xLarge = FontSize(21.0);
 static const xxLarge = FontSize(28.0);
 static const smaller = FontSize(-0.83);
 static const larger = FontSize(-1.2);
}

class LineHeight {
 final double? size;
 final String units;

 const LineHeight(this.size, {this.units = ""});

 factory LineHeight.percent(double percent) {
   return LineHeight(percent / 100.0 * 1.2, units: "%");
 }

 factory LineHeight.em(double em) {
   return LineHeight(em * 1.2, units: "em");
 }

 factory LineHeight.rem(double rem) {
   return LineHeight(rem * 1.2, units: "rem");
 }
```

```
 factory LineHeight.number(double num) {
   return LineHeight(num * 1.2, units: "number");
 }

 static const normal = LineHeight(1.2);
}

class ListStyleType {
 final String text;
 final String type;
 final Widget? widget;

 const ListStyleType(this.text, {this.type = "marker", this.widget});

 factory ListStyleType.fromImage(String url) => ListStyleType(url, type:
"image");

 factory ListStyleType.fromWidget(Widget widget) => ListStyleType("", widget:
widget, type: "widget");

 static const LOWER_ALPHA = ListStyleType("LOWER_ALPHA");
 static const UPPER_ALPHA = ListStyleType("UPPER_ALPHA");
 static const LOWER_LATIN = ListStyleType("LOWER_LATIN");
 static const UPPER_LATIN = ListStyleType("UPPER_LATIN");
 static const CIRCLE = ListStyleType("CIRCLE");
 static const DISC = ListStyleType("DISC");
 static const DECIMAL = ListStyleType("DECIMAL");
 static const LOWER_ROMAN = ListStyleType("LOWER_ROMAN");
 static const UPPER_ROMAN = ListStyleType("UPPER_ROMAN");
 static const SQUARE = ListStyleType("SQUARE");
 static const NONE = ListStyleType("NONE");
}

enum ListStylePosition {
 OUTSIDE,
 INSIDE,
}

enum TextTransform {
 uppercase,
 lowercase,
 capitalize,
 none,
}

enum VerticalAlign {
 BASELINE,
 SUB,
 SUPER,
}

enum WhiteSpace {
```

```
 NORMAL,
 PRE,
}
```

Html_parser.dart –

```
import 'dart:collection';
import 'dart:math';

import 'package:collection/collection.dart';
import 'package:csslib/parser.dart' as cssparser;
import 'package:csslib/visitor.dart' as css;
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';
import 'package:flutter/rendering.dart';
import 'package:flutter_html/flutter_html.dart';
import 'package:flutter_html/image_render.dart';
import 'package:flutter_html/src/anchor.dart';
import 'package:flutter_html/src/css_parser.dart';
import 'package:flutter_html/src/html_elements.dart';
import 'package:flutter_html/src/layout_element.dart';
import 'package:flutter_html/src/navigation_delegate.dart';
import 'package:flutter_html/src/utils.dart';
import 'package:flutter_html/style.dart';
import 'package:html/dom.dart' as dom;
import 'package:html/parser.dart' as htmlparser;
import 'package:numerus/numerus.dart';

typedef OnTap = void Function(
    String? url,
    RenderContext context,
    Map<String, String> attributes,
    dom.Element? element,
);
typedef OnMathError = Widget Function(
    String parsedTex,
    String exception,
    String exceptionWithType,
);
typedef OnCssParseError = String? Function(
  String css,
  List<cssparser.Message> errors,
);
typedef CustomRender = dynamic Function(
  RenderContext context,
  Widget parsedChild,
);
```

```
class HtmlParser extends StatelessWidget {
  final Key? key;
  final dom.Document htmlData;
  final OnTap? onLinkTap;
  final OnTap? onAnchorTap;
  final OnTap? onImageTap;
  final OnCssParseError? onCssParseError;
  final ImageErrorListener? onImageError;
  final OnMathError? onMathError;
  final bool shrinkWrap;
  final bool selectable;

  final Map<String, Style> style;
  final Map<String, CustomRender> customRender;
  final Map<ImageSourceMatcher, ImageRender> imageRenders;
  final List<String> tagsList;
  final NavigationDelegate? navigationDelegateForIframe;
  final OnTap? _onAnchorTap;
  final TextSelectionControls? selectionControls;
  final ScrollPhysics? scrollPhysics;

  final Map<String, Size> cachedImageSizes = {};

  HtmlParser({
    required this.key,
    required this.htmlData,
    required this.onLinkTap,
    required this.onAnchorTap,
    required this.onImageTap,
    required this.onCssParseError,
    required this.onImageError,
    required this.onMathError,
    required this.shrinkWrap,
    required this.selectable,
    required this.style,
    required this.customRender,
    required this.imageRenders,
    required this.tagsList,
    required this.navigationDelegateForIframe,
    this.selectionControls,
    this.scrollPhysics,
  })  : this._onAnchorTap = onAnchorTap != null
          ? onAnchorTap
          : key != null
              ? _handleAnchorTap(key, onLinkTap)
              : null,
        super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    Map<String, Map<String, List<css.Expression>>> declarations =
_getExternalCssDeclarations(htmlData.getElementsByTagName("style"), onCssParseError);
    StyledElement lexedTree = lexDomTree(
      htmlData,
      customRender.keys.toList(),
      tagsList,
      navigationDelegateForIframe,
      context,
    );
    StyledElement? externalCssStyledTree;
    if (declarations.isNotEmpty) {
      externalCssStyledTree = _applyExternalCss(declarations, lexedTree);
    }
    StyledElement inlineStyledTree = _applyInlineStyles(externalCssStyledTree ?? lexedTree,
onCssParseError);
    StyledElement customStyledTree = _applyCustomStyles(style, inlineStyledTree);
    StyledElement cascadedStyledTree = _cascadeStyles(style, customStyledTree);
    StyledElement cleanedTree = cleanTree(cascadedStyledTree);
    InlineSpan parsedTree = parseTree(
      RenderContext(
        buildContext: context,
        parser: this,
        tree: cleanedTree,
        style: cleanedTree.style,
      ),
      cleanedTree,
    );

    // This is the final scaling that assumes any other StyledText instances are
    // using textScaleFactor = 1.0 (which is the default). This ensures the correct
    // scaling is used, but relies on https://github.com/flutter/flutter/pull/59711
    // to wrap everything when larger accessibility fonts are used.
    if (selectable) {
      return StyledText.selectable(
        textSpan: parsedTree as TextSpan,
        style: cleanedTree.style,
        textScaleFactor: MediaQuery.of(context).textScaleFactor,
        renderContext: RenderContext(
          buildContext: context,
          parser: this,
          tree: cleanedTree,
          style: cleanedTree.style,
        ),
        selectionControls: selectionControls,
        scrollPhysics: scrollPhysics,
      );
```

```
  }
  return StyledText(
    textSpan: parsedTree,
    style: cleanedTree.style,
    textScaleFactor: MediaQuery.of(context).textScaleFactor,
    renderContext: RenderContext(
      buildContext: context,
      parser: this,
      tree: cleanedTree,
      style: cleanedTree.style,
    ),
  );
}

/// [parseHTML] converts a string of HTML to a DOM document using the dart `html` library.
static dom.Document parseHTML(String data) {
  return htmlparser.parse(data);
}

/// [parseCss] converts a string of CSS to a CSS stylesheet using the dart `csslib` library.
static css.StyleSheet parseCss(String data) {
  return cssparser.parse(data);
}

/// [lexDomTree] converts a DOM document to a simplified tree of [StyledElement]s.
static StyledElement lexDomTree(
  dom.Document html,
  List<String> customRenderTags,
  List<String> tagsList,
  NavigationDelegate? navigationDelegateForIframe,
  BuildContext context,
) {
  StyledElement tree = StyledElement(
    name: "[Tree Root]",
    children: <StyledElement>[],
    node: html.documentElement,
    style: Style.fromTextStyle(Theme.of(context).textTheme.bodyMedium!),
  );

  html.nodes.forEach((node) {
    tree.children.add(_recursiveLexer(
      node,
      customRenderTags,
      tagsList,
      navigationDelegateForIframe,
    ));
  });
```

```
  return tree;
}

/// [_recursiveLexer] is the recursive worker function for [lexDomTree].
///
/// It runs the parse functions of every type of
/// element and returns a [StyledElement] tree representing the element.
static StyledElement _recursiveLexer(
  dom.Node node,
  List<String> customRenderTags,
  List<String> tagsList,
  NavigationDelegate? navigationDelegateForIframe,
) {
  List<StyledElement> children = <StyledElement>[];

  node.nodes.forEach((childNode) {
    children.add(_recursiveLexer(
      childNode,
      customRenderTags,
      tagsList,
      navigationDelegateForIframe,
    ));
  });

  //TODO(Sub6Resources): There's probably a more efficient way to look this up.
  if (node is dom.Element) {
    if (!tagsList.contains(node.localName)) {
      return EmptyContentElement();
    }
    if (STYLED_ELEMENTS.contains(node.localName)) {
      return parseStyledElement(node, children);
    } else if (INTERACTABLE_ELEMENTS.contains(node.localName)) {
      return parseInteractableElement(node, children);
    } else if (REPLACED_ELEMENTS.contains(node.localName)) {
      return parseReplacedElement(node, children, navigationDelegateForIframe);
    } else if (LAYOUT_ELEMENTS.contains(node.localName)) {
      return parseLayoutElement(node, children);
    } else if (TABLE_CELL_ELEMENTS.contains(node.localName)) {
      return parseTableCellElement(node, children);
    } else if (TABLE_DEFINITION_ELEMENTS.contains(node.localName)) {
      return parseTableDefinitionElement(node, children);
    } else if (customRenderTags.contains(node.localName)) {
      return parseStyledElement(node, children);
    } else {
      return EmptyContentElement();
    }
  } else if (node is dom.Text) {
```

```
      return TextContentElement(text: node.text, style: Style(), element: node.parent, node:
node);
    } else {
      return EmptyContentElement();
    }
  }

  static Map<String, Map<String, List<css.Expression>>>
_getExternalCssDeclarations(List<dom.Element> styles, OnCssParseError? errorHandler) {
    String fullCss = "";
    for (final e in styles) {
      fullCss = fullCss + e.innerHtml;
    }
    if (fullCss.isNotEmpty) {
      final declarations = parseExternalCss(fullCss, errorHandler);
      return declarations;
    } else {
      return {};
    }
  }

  static StyledElement _applyExternalCss(Map<String, Map<String, List<css.Expression>>>
declarations, StyledElement tree) {
    declarations.forEach((key, style) {
      try {
        if (tree.matchesSelector(key)) {
          tree.style = tree.style.merge(declarationsToStyle(style));
        }
      } catch (_) {}
    });

    tree.children.forEach((e) => _applyExternalCss(declarations, e));

    return tree;
  }

  static StyledElement _applyInlineStyles(StyledElement tree, OnCssParseError?
errorHandler) {
    if (tree.attributes.containsKey("style")) {
      final newStyle = inlineCssToStyle(tree.attributes['style'], errorHandler);
      if (newStyle != null) {
        tree.style = tree.style.merge(newStyle);
      }
    }

    tree.children.forEach((e) => _applyInlineStyles(e, errorHandler));
    return tree;
  }
```

```
  /// [applyCustomStyles] applies the [Style] objects passed into the [Html]
  /// widget onto the [StyledElement] tree, no cascading of styles is done at this point.
  static StyledElement _applyCustomStyles(Map<String, Style> style, StyledElement tree) {
    style.forEach((key, style) {
      try {
        if (tree.matchesSelector(key)) {
          tree.style = tree.style.merge(style);
        }
      } catch (_) {}
    });
    tree.children.forEach((e) => _applyCustomStyles(style, e));

    return tree;
  }

  /// [_cascadeStyles] cascades all of the inherited styles down the tree, applying them to
each
  /// child that doesn't specify a different style.
  static StyledElement _cascadeStyles(Map<String, Style> style, StyledElement tree) {
    tree.children.forEach((child) {
      child.style = tree.style.copyOnlyInherited(child.style);
      _cascadeStyles(style, child);
    });

    return tree;
  }

  /// [cleanTree] optimizes the [StyledElement] tree so all [BlockElement]s are
  /// on the first level, redundant levels are collapsed, empty elements are
  /// removed, and specialty elements are processed.
  static StyledElement cleanTree(StyledElement tree) {
    tree = _processInternalWhitespace(tree);
    tree = _processInlineWhitespace(tree);
    tree = _removeEmptyElements(tree);
    tree = _processListCharacters(tree);
    tree = _processBeforesAndAfters(tree);
    tree = _collapseMargins(tree);
    tree = _processFontSize(tree);
    return tree;
  }

  /// [parseTree] converts a tree of [StyledElement]s to an [InlineSpan] tree.
  ///
  /// [parseTree] is responsible for handling the [customRender] parameter and
  /// deciding what different `Style.display` options look like as Widgets.
  InlineSpan parseTree(RenderContext context, StyledElement tree) {
    // Merge this element's style into the context so that children
```

```
// inherit the correct style
RenderContext newContext = RenderContext(
  buildContext: context.buildContext,
  parser: this,
  tree: tree,
  style: context.style.copyOnlyInherited(tree.style),
);

if (customRender.containsKey(tree.name)) {
  final render = customRender[tree.name]!.call(
    newContext,
    ContainerSpan(
      key: AnchorKey.of(key, tree),
      newContext: newContext,
      style: tree.style,
      shrinkWrap: context.parser.shrinkWrap,
      children: tree.children.map((tree) => parseTree(newContext, tree)).toList(),
    ),
  );
  if (render != null) {
    assert(render is InlineSpan || render is Widget);
    return render is InlineSpan
        ? render
        : WidgetSpan(
            child: ContainerSpan(
              key: AnchorKey.of(key, tree),
              newContext: newContext,
              style: tree.style,
              shrinkWrap: context.parser.shrinkWrap,
              child: render,
            ),
          );
  }
}

//Return the correct InlineSpan based on the element type.
if (tree.style.display == Display.BLOCK &&
    (tree.children.isNotEmpty || tree.element?.localName == "hr")) {
  if (newContext.parser.selectable) {
    return TextSpan(
      style: newContext.style.generateTextStyle(),
      children: tree.children
          .expandIndexed((i, childTree) => [
        if (childTree.style.display == Display.BLOCK &&
            i > 0 &&
            tree.children[i - 1] is ReplacedElement)
          TextSpan(text: "\n"),
        parseTree(newContext, childTree),
```

```
          if (i != tree.children.length - 1 &&
              childTree.style.display == Display.BLOCK &&
              childTree.element?.localName != "html" &&
              childTree.element?.localName != "body")
            TextSpan(text: "\n"),
        ])
            .toList(),
      );
    }
    return WidgetSpan(
      child: ContainerSpan(
        key: AnchorKey.of(key, tree),
        newContext: newContext,
        style: tree.style,
        shrinkWrap: context.parser.shrinkWrap,
        children: tree.children
            .expandIndexed((i, childTree) => [
                  if (shrinkWrap &&
                      childTree.style.display == Display.BLOCK &&
                      i > 0 &&
                      tree.children[i - 1] is ReplacedElement)
                    TextSpan(text: "\n"),
                  parseTree(newContext, childTree),
                  if (shrinkWrap &&
                      i != tree.children.length - 1 &&
                      childTree.style.display == Display.BLOCK &&
                      childTree.element?.localName != "html" &&
                      childTree.element?.localName != "body")
                    TextSpan(text: "\n"),
                ])
            .toList(),
      ),
    );
  } else if (tree.style.display == Display.LIST_ITEM) {
    List<InlineSpan> getChildren(StyledElement tree) {
      List<InlineSpan> children = tree.children.map((tree) => parseTree(newContext,
tree)).toList();
      if (tree.style.listStylePosition == ListStylePosition.INSIDE) {
        final tabSpan = WidgetSpan(
          child: Text("\t", textAlign: TextAlign.right, style: TextStyle(fontWeight:
FontWeight.w400)),
        );
        children.insert(0, tabSpan);
      }
      return children;
    }

    return WidgetSpan(
```

```
        child: ContainerSpan(
          key: AnchorKey.of(key, tree),
          newContext: newContext,
          style: tree.style,
          shrinkWrap: context.parser.shrinkWrap,
          child: Row(
            crossAxisAlignment: CrossAxisAlignment.start,
            mainAxisSize: MainAxisSize.min,
            textDirection: tree.style.direction,
            children: [
              tree.style.listStylePosition == ListStylePosition.OUTSIDE ?
              Padding(
                padding: tree.style.padding?.nonNegative ?? EdgeInsets.only(left:
tree.style.direction != TextDirection.rtl ? 10.0 : 0.0, right: tree.style.direction ==
TextDirection.rtl ? 10.0 : 0.0),
                child: newContext.style.markerContent
              ) : Container(height: 0, width: 0),
              Text("\t", textAlign: TextAlign.right, style: TextStyle(fontWeight: FontWeight.w400)),
              Expanded(
                child: Padding(
                  padding: tree.style.listStylePosition == ListStylePosition.INSIDE ?
                    EdgeInsets.only(left: tree.style.direction != TextDirection.rtl ? 10.0 : 0.0, right:
tree.style.direction == TextDirection.rtl ? 10.0 : 0.0) : EdgeInsets.zero,
                  child: StyledText(
                    textSpan: TextSpan(
                      children: getChildren(tree)..insertAll(0, tree.style.listStylePosition ==
ListStylePosition.INSIDE ?
                        [
                          WidgetSpan(alignment: PlaceholderAlignment.middle, child:
newContext.style.markerContent ?? Container(height: 0, width: 0))
                        ] : []),
                      style: newContext.style.generateTextStyle(),
                    ),
                    style: newContext.style,
                    renderContext: context,
                  )
                )
              )
            ],
          ),
        ),
      );
    } else if (tree is ReplacedElement) {
      if (tree is TextContentElement) {
        return TextSpan(text: tree.text?.transformed(tree.style.textTransform));
      } else {
        return WidgetSpan(
          alignment: tree.alignment,
```

```
          baseline: TextBaseline.alphabetic,
          child: tree.toWidget(newContext)!,
        );
      }
    } else if (tree is InteractableElement) {
      InlineSpan addTaps(InlineSpan childSpan, TextStyle childStyle) {
        if (childSpan is TextSpan) {
          return TextSpan(
            mouseCursor: SystemMouseCursors.click,
            text: childSpan.text,
            children: childSpan.children
                ?.map((e) => addTaps(e, childStyle.merge(childSpan.style)))
                .toList(),
            style: newContext.style.generateTextStyle().merge(
                childSpan.style == null
                    ? childStyle
                    : childStyle.merge(childSpan.style)),
            semanticsLabel: childSpan.semanticsLabel,
            recognizer: TapGestureRecognizer()
              ..onTap =
                  _onAnchorTap != null ? () => _onAnchorTap!(tree.href, context, tree.attributes,
tree.element) : null,
          );
        } else {
          return WidgetSpan(
            child: MouseRegion(
              key: AnchorKey.of(key, tree),
              cursor: SystemMouseCursors.click,
              child: MultipleTapGestureDetector(
                onTap: _onAnchorTap != null
                ? () => _onAnchorTap!(tree.href, context, tree.attributes, tree.element)
                    : null,
                child: GestureDetector(
                  key: AnchorKey.of(key, tree),
                  onTap: _onAnchorTap != null
                  ? () => _onAnchorTap!(tree.href, context, tree.attributes, tree.element)
                      : null,
                  child: (childSpan as WidgetSpan).child,
                ),
              ),
            ),
          );
        }
      }

      return TextSpan(
        mouseCursor: SystemMouseCursors.click,
        children: tree.children
```

```
            .map((tree) => parseTree(newContext, tree))
            .map((childSpan) {
        return addTaps(childSpan,
          newContext.style.generateTextStyle().merge(childSpan.style));
        }).toList(),
    );
  } else if (tree is LayoutElement) {
    return WidgetSpan(
      child: tree.toWidget(context)!,
    );
  } else if (tree.style.verticalAlign != null &&
      tree.style.verticalAlign != VerticalAlign.BASELINE) {
    late double verticalOffset;
    switch (tree.style.verticalAlign) {
      case VerticalAlign.SUB:
        verticalOffset = tree.style.fontSize!.size! / 2.5;
        break;
      case VerticalAlign.SUPER:
        verticalOffset = tree.style.fontSize!.size! / -2.5;
        break;
      default:
        break;
    }
    //Requires special layout features not available in the TextStyle API.
    return WidgetSpan(
      child: Transform.translate(
        key: AnchorKey.of(key, tree),
        offset: Offset(0, verticalOffset),
        child: StyledText(
          textSpan: TextSpan(
            style: newContext.style.generateTextStyle(),
            children: tree.children.map((tree) => parseTree(newContext, tree)).toList(),
          ),
          style: newContext.style,
          renderContext: newContext,
        ),
      ),
    );
  } else {
    ///[tree] is an inline element.
    return TextSpan(
      style: newContext.style.generateTextStyle(),
      children: tree.children
          .expand((tree) => [
                parseTree(newContext, tree),
                if (tree.style.display == Display.BLOCK &&
                    tree.element?.localName != "html" &&
                    tree.element?.localName != "body")
```

```
                TextSpan(text: "\n"),
              ])
          .toList(),
    );
  }
 }

 static OnTap _handleAnchorTap(Key key, OnTap? onLinkTap) =>
      (String? url, RenderContext context, Map<String, String> attributes, dom.Element?
element) {
      if (url?.startsWith("#") == true) {
        final anchorContext = AnchorKey.forId(key, url!.substring(1))?.currentContext;
        if (anchorContext != null) {
          Scrollable.ensureVisible(anchorContext);
        }
        return;
      }
      onLinkTap?.call(url, context, attributes, element);
    };

 /// [processWhitespace] removes unnecessary whitespace from the StyledElement tree.
 ///
 /// The criteria for determining which whitespace is replaceable is outlined
 /// at https://www.w3.org/TR/css-text-3/
 /// and summarized at
https://medium.com/@patrickbrosset/when-does-white-space-matter-in-html-b90e8a7cdd33
 static StyledElement _processInternalWhitespace(StyledElement tree) {
   if ((tree.style.whiteSpace ?? WhiteSpace.NORMAL) == WhiteSpace.PRE) {
     // Preserve this whitespace
   } else if (tree is TextContentElement) {
     tree.text = _removeUnnecessaryWhitespace(tree.text!);
   } else {
     tree.children.forEach(_processInternalWhitespace);
   }
   return tree;
 }

 /// [_processInlineWhitespace] is responsible for removing redundant whitespace
 /// between and among inline elements. It does so by creating a boolean [Context]
 /// and passing it to the [_processInlineWhitespaceRecursive] function.
 static StyledElement _processInlineWhitespace(StyledElement tree) {
   tree = _processInlineWhitespaceRecursive(tree, Context(false));
   return tree;
 }

 /// [_processInlineWhitespaceRecursive] analyzes the whitespace between and among
different
```

```
  /// inline elements, and replaces any instance of two or more spaces with a single space,
according
  /// to the w3's HTML whitespace processing specification linked to above.
  static StyledElement _processInlineWhitespaceRecursive(
    StyledElement tree,
    Context<bool> keepLeadingSpace,
  ) {
    if (tree is TextContentElement) {
      /// initialize indices to negative numbers to make conditionals a little easier
      int textIndex = -1;
      int elementIndex = -1;
      /// initialize parent after to a whitespace to account for elements that are
      /// the last child in the list of elements
      String parentAfterText = " ";
      /// find the index of the text in the current tree
      if ((tree.element?.nodes.length ?? 0) >= 1) {
        textIndex = tree.element?.nodes.indexWhere((element) => element == tree.node) ??
-1;
      }
      /// get the parent nodes
      dom.NodeList? parentNodes = tree.element?.parent?.nodes;
      /// find the index of the tree itself in the parent nodes
      if ((parentNodes?.length ?? 0) >= 1) {
        elementIndex = parentNodes?.indexWhere((element) => element == tree.element) ??
-1;
      }
      /// if the tree is any node except the last node in the node list and the
      /// next node in the node list is a text node, then get its text. Otherwise
      /// the next node will be a [dom.Element], so keep unwrapping that until
      /// we get the underlying text node, and finally get its text.
      if (elementIndex < (parentNodes?.length ?? 1) - 1 && parentNodes?[elementIndex + 1] is
dom.Text) {
        parentAfterText = parentNodes?[elementIndex + 1].text ?? " ";
      } else if (elementIndex < (parentNodes?.length ?? 1) - 1) {
        var parentAfter = parentNodes?[elementIndex + 1];
        while (parentAfter is dom.Element) {
          if (parentAfter.nodes.isNotEmpty) {
            parentAfter = parentAfter.nodes.first;
          } else {
            break;
          }
        }
        parentAfterText = parentAfter?.text ?? " ";
      }
      /// If the text is the first element in the current tree node list, it
      /// starts with a whitespace, it isn't a line break, either the
      /// whitespace is unnecessary or it is a block element, and either it is
      /// first element in the parent node list or the previous element
```

```
    /// in the parent node list ends with a whitespace, delete it.
    ///
    /// We should also delete the whitespace at any point in the node list
    /// if the previous element is a <br> because that tag makes the element
    /// act like a block element.
    if (textIndex < 1
        && tree.text!.startsWith(' ')
        && tree.element?.localName != "br"
        && (!keepLeadingSpace.data
            || BLOCK_ELEMENTS.contains(tree.element?.localName ?? ""))
        && (elementIndex < 1
            || (elementIndex >= 1
                && parentNodes?[elementIndex - 1] is dom.Text
                && parentNodes![elementIndex - 1].text!.endsWith(" ")))
    ) {
      tree.text = tree.text!.replaceFirst(' ', '');
    } else if (textIndex >= 1
        && tree.text!.startsWith(' ')
        && tree.element?.nodes[textIndex - 1] is dom.Element
        && (tree.element?.nodes[textIndex - 1] as dom.Element).localName == "br"
    ) {
      tree.text = tree.text!.replaceFirst(' ', '');
    }
    /// If the text is the last element in the current tree node list, it isn't
    /// a line break, and the next text node starts with a whitespace,
    /// update the [Context] to signify to that next text node whether it should
    /// keep its whitespace. This is based on whether the current text ends with a
    /// whitespace.
    if (textIndex == (tree.element?.nodes.length ?? 1) - 1
        && tree.element?.localName != "br"
        && parentAfterText.startsWith(' ')
    ) {
      keepLeadingSpace.data = !tree.text!.endsWith(' ');
    }
  }

  tree.children.forEach((e) => _processInlineWhitespaceRecursive(e, keepLeadingSpace));

  return tree;
}

/// [removeUnnecessaryWhitespace] removes "unnecessary" white space from the given
String.
///
/// The steps for removing this whitespace are as follows:
/// (1) Remove any whitespace immediately preceding or following a newline.
/// (2) Replace all newlines with a space
/// (3) Replace all tabs with a space
```

```
  /// (4) Replace any instances of two or more spaces with a single space.
  static String _removeUnnecessaryWhitespace(String text) {
    return text
        .replaceAll(RegExp("\ *(?=\n)"), "\n")
        .replaceAll(RegExp("(?:\n)\ *"), "\n")
        .replaceAll("\n", " ")
        .replaceAll("\t", " ")
        .replaceAll(RegExp(" {2,}"), " ");
  }

  /// [processListCharacters] adds list characters to the front of all list items.
  ///
  /// The function uses the [_processListCharactersRecursive] function to do most of its work.
  static StyledElement _processListCharacters(StyledElement tree) {
    final olStack = ListQueue<Context>();
    tree = _processListCharactersRecursive(tree, olStack);
    return tree;
  }

  /// [_processListCharactersRecursive] uses a Stack of integers to properly number and
  /// bullet all list items according to the [ListStyleType] they have been given.
  static StyledElement _processListCharactersRecursive(
      StyledElement tree, ListQueue<Context> olStack) {
    if (tree.style.listStylePosition == null) {
      tree.style.listStylePosition = ListStylePosition.OUTSIDE;
    }
    if (tree.name == 'ol' && tree.style.listStyleType != null && tree.style.listStyleType!.type ==
"marker") {
      switch (tree.style.listStyleType!) {
        case ListStyleType.LOWER_LATIN:
        case ListStyleType.LOWER_ALPHA:
        case ListStyleType.UPPER_LATIN:
        case ListStyleType.UPPER_ALPHA:
          olStack.add(Context<String>('a'));
          if ((tree.attributes['start'] != null ? int.tryParse(tree.attributes['start']!) : null) != null) {
            var start = int.tryParse(tree.attributes['start']!) ?? 1;
            var x = 1;
            while (x < start) {
              olStack.last.data = olStack.last.data.toString().nextLetter();
              x++;
            }
          }
          break;
        default:
          olStack.add(Context<int>((tree.attributes['start'] != null ?
int.tryParse(tree.attributes['start'] ?? "") ?? 1 : 1) - 1));
          break;
      }
```

```
    } else if (tree.style.display == Display.LIST_ITEM && tree.style.listStyleType != null &&
tree.style.listStyleType!.type == "widget") {
      tree.style.markerContent = tree.style.listStyleType!.widget!;
    } else if (tree.style.display == Display.LIST_ITEM && tree.style.listStyleType != null &&
tree.style.listStyleType!.type == "image") {
      tree.style.markerContent = Image.network(tree.style.listStyleType!.text);
    } else if (tree.style.display == Display.LIST_ITEM && tree.style.listStyleType != null) {
      String marker = "";
      switch (tree.style.listStyleType!) {
        case ListStyleType.NONE:
          break;
        case ListStyleType.CIRCLE:
          marker = '○';
          break;
        case ListStyleType.SQUARE:
          marker = '■';
          break;
        case ListStyleType.DISC:
          marker = '•';
          break;
        case ListStyleType.DECIMAL:
          if (olStack.isEmpty) {
            olStack.add(Context<int>((tree.attributes['start'] != null ?
int.tryParse(tree.attributes['start'] ?? "") ?? 1 : 1) - 1));
          }
          olStack.last.data += 1;
          marker = '${olStack.last.data}.';
          break;
        case ListStyleType.LOWER_LATIN:
        case ListStyleType.LOWER_ALPHA:
          if (olStack.isEmpty) {
            olStack.add(Context<String>('a'));
            if ((tree.attributes['start'] != null ? int.tryParse(tree.attributes['start']!) : null) != null) {
              var start = int.tryParse(tree.attributes['start']!) ?? 1;
              var x = 1;
              while (x < start) {
                olStack.last.data = olStack.last.data.toString().nextLetter();
                x++;
              }
            }
          }
          marker = olStack.last.data.toString() + ".";
          olStack.last.data = olStack.last.data.toString().nextLetter();
          break;
        case ListStyleType.UPPER_LATIN:
        case ListStyleType.UPPER_ALPHA:
          if (olStack.isEmpty) {
            olStack.add(Context<String>('a'));
```

```
          if ((tree.attributes['start'] != null ? int.tryParse(tree.attributes['start']!) : null) != null) {
            var start = int.tryParse(tree.attributes['start']!) ?? 1;
            var x = 1;
            while (x < start) {
              olStack.last.data = olStack.last.data.toString().nextLetter();
              x++;
            }
          }
        }
        marker = olStack.last.data.toString().toUpperCase() + ".";
        olStack.last.data = olStack.last.data.toString().nextLetter();
        break;
      case ListStyleType.LOWER_ROMAN:
        if (olStack.isEmpty) {
          olStack.add(Context<int>((tree.attributes['start'] != null ?
int.tryParse(tree.attributes['start'] ?? "") ?? 1 : 1) - 1));
        }
        olStack.last.data += 1;
        if (olStack.last.data <= 0) {
          marker = '${olStack.last.data}.';
        } else {
          marker = (olStack.last.data as int).toRomanNumeralString()!.toLowerCase() + ".";
        }
        break;
      case ListStyleType.UPPER_ROMAN:
        if (olStack.isEmpty) {
          olStack.add(Context<int>((tree.attributes['start'] != null ?
int.tryParse(tree.attributes['start'] ?? "") ?? 1 : 1) - 1));
        }
        olStack.last.data += 1;
        if (olStack.last.data <= 0) {
          marker = '${olStack.last.data}.';
        } else {
          marker = (olStack.last.data as int).toRomanNumeralString()! + ".";
        }
        break;
    }
    tree.style.markerContent = Text(
        marker,
        textAlign: TextAlign.right,
    );
  }

  tree.children.forEach((e) => _processListCharactersRecursive(e, olStack));

  if (tree.name == 'ol') {
    olStack.removeLast();
  }
```

```
    return tree;
  }

  /// [_processBeforesAndAfters] adds text content to the beginning and end of
  /// the list of the trees children according to the `before` and `after` Style
  /// properties.
  static StyledElement _processBeforesAndAfters(StyledElement tree) {
    if (tree.style.before != null) {
      tree.children.insert(
          0, TextContentElement(text: tree.style.before, style:
tree.style.copyWith(beforeAfterNull: true, display: Display.INLINE)));
    }
    if (tree.style.after != null) {
      tree.children
          .add(TextContentElement(text: tree.style.after, style:
tree.style.copyWith(beforeAfterNull: true, display: Display.INLINE)));
    }

    tree.children.forEach(_processBeforesAndAfters);

    return tree;
  }

  /// [collapseMargins] follows the specifications at
https://www.w3.org/TR/CSS21/box.html#collapsing-margins
  /// for collapsing margins of block-level boxes. This prevents the doubling of margins
between
  /// boxes, and makes for a more correct rendering of the html content.
  ///
  /// Paraphrased from the CSS specification:
  /// Margins are collapsed if both belong to vertically-adjacent box edges, i.e form one of the
following pairs:
  /// (1) Top margin of a box and top margin of its first in-flow child
  /// (2) Bottom margin of a box and top margin of its next in-flow following sibling
  /// (3) Bottom margin of a last in-flow child and bottom margin of its parent (if the parent's
height is not explicit)
  /// (4) Top and Bottom margins of a box with a height of zero or no in-flow children.
  static StyledElement _collapseMargins(StyledElement tree) {
    //Short circuit if we've reached a leaf of the tree
    if (tree.children.isEmpty) {
      // Handle case (4) from above.
      if ((tree.style.height ?? 0) == 0) {
        tree.style.margin = EdgeInsets.zero;
      }
      return tree;
    }
```

```
//Collapsing should be depth-first.
tree.children.forEach(_collapseMargins);

//The root boxes do not collapse.
if (tree.name == '[Tree Root]' || tree.name == 'html') {
  return tree;
}

// Handle case (1) from above.
// Top margins cannot collapse if the element has padding
if ((tree.style.padding?.top ?? 0) == 0) {
  final parentTop = tree.style.margin?.top ?? 0;
  final firstChildTop = tree.children.first.style.margin?.top ?? 0;
  final newOuterMarginTop = max(parentTop, firstChildTop);

  // Set the parent's margin
  if (tree.style.margin == null) {
    tree.style.margin = EdgeInsets.only(top: newOuterMarginTop);
  } else {
    tree.style.margin = tree.style.margin!.copyWith(top: newOuterMarginTop);
  }

  // And remove the child's margin
  if (tree.children.first.style.margin == null) {
    tree.children.first.style.margin = EdgeInsets.zero;
  } else {
    tree.children.first.style.margin =
        tree.children.first.style.margin!.copyWith(top: 0);
  }
}

// Handle case (3) from above.
// Bottom margins cannot collapse if the element has padding
if ((tree.style.padding?.bottom ?? 0) == 0) {
  final parentBottom = tree.style.margin?.bottom ?? 0;
  final lastChildBottom = tree.children.last.style.margin?.bottom ?? 0;
  final newOuterMarginBottom = max(parentBottom, lastChildBottom);

  // Set the parent's margin
  if (tree.style.margin == null) {
    tree.style.margin = EdgeInsets.only(bottom: newOuterMarginBottom);
  } else {
    tree.style.margin =
        tree.style.margin!.copyWith(bottom: newOuterMarginBottom);
  }

  // And remove the child's margin
  if (tree.children.last.style.margin == null) {
```

```
      tree.children.last.style.margin = EdgeInsets.zero;
    } else {
      tree.children.last.style.margin =
          tree.children.last.style.margin!.copyWith(bottom: 0);
    }
  }

  // Handle case (2) from above.
  if (tree.children.length > 1) {
    for (int i = 1; i < tree.children.length; i++) {
      final previousSiblingBottom =
          tree.children[i - 1].style.margin?.bottom ?? 0;
      final thisTop = tree.children[i].style.margin?.top ?? 0;
      final newInternalMargin = max(previousSiblingBottom, thisTop) / 2;

      if (tree.children[i - 1].style.margin == null) {
        tree.children[i - 1].style.margin =
            EdgeInsets.only(bottom: newInternalMargin);
      } else {
        tree.children[i - 1].style.margin = tree.children[i - 1].style.margin!
            .copyWith(bottom: newInternalMargin);
      }

      if (tree.children[i].style.margin == null) {
        tree.children[i].style.margin =
            EdgeInsets.only(top: newInternalMargin);
      } else {
        tree.children[i].style.margin =
            tree.children[i].style.margin!.copyWith(top: newInternalMargin);
      }
    }
  }

  return tree;
}

/// [removeEmptyElements] recursively removes empty elements.
///
/// An empty element is any [EmptyContentElement], any empty [TextContentElement],
/// or any block-level [TextContentElement] that contains only whitespace and doesn't follow
/// a block element or a line break.
static StyledElement _removeEmptyElements(StyledElement tree) {
  List<StyledElement> toRemove = <StyledElement>[];
  bool lastChildBlock = true;
  tree.children.forEach((child) {
    if (child is EmptyContentElement || child is EmptyLayoutElement) {
      toRemove.add(child);
    } else if (child is TextContentElement
```

```
          && (tree.name == "body" || tree.name == "ul")
          && child.text!.replaceAll(' ', '').isEmpty) {
        toRemove.add(child);
      } else if (child is TextContentElement
          && child.text!.isEmpty
          && child.style.whiteSpace != WhiteSpace.PRE) {
        toRemove.add(child);
      } else if (child is TextContentElement &&
          child.style.whiteSpace != WhiteSpace.PRE &&
          tree.style.display == Display.BLOCK &&
          child.text!.isEmpty &&
          lastChildBlock) {
        toRemove.add(child);
      } else if (child.style.display == Display.NONE) {
        toRemove.add(child);
      } else {
        _removeEmptyElements(child);
      }

      // This is used above to check if the previous element is a block element or a line break.
      lastChildBlock = (child.style.display == Display.BLOCK ||
          child.style.display == Display.LIST_ITEM ||
          (child is TextContentElement && child.text == '\n'));
    });
    tree.children.removeWhere((element) => toRemove.contains(element));

    return tree;
  }

  /// [_processFontSize] changes percent-based font sizes (negative numbers in this
implementation)
  /// to pixel-based font sizes.
  static StyledElement _processFontSize(StyledElement tree) {
    double? parentFontSize = tree.style.fontSize?.size ?? FontSize.medium.size;

    tree.children.forEach((child) {
      if ((child.style.fontSize?.size ?? parentFontSize)! < 0) {
        child.style.fontSize =
            FontSize(parentFontSize! * -child.style.fontSize!.size!);
      }

      _processFontSize(child);
    });
    return tree;
  }
}

/// The [RenderContext] is available when parsing the tree. It contains information
```

```
/// about the [BuildContext] of the `Html` widget, contains the configuration available
/// in the [HtmlParser], and contains information about the [Style] of the current
/// tree root.
class RenderContext {
  final BuildContext buildContext;
  final HtmlParser parser;
  final StyledElement tree;
  final Style style;

  RenderContext({
    required this.buildContext,
    required this.parser,
    required this.tree,
    required this.style,
  });
}

/// A [ContainerSpan] is a widget with an [InlineSpan] child or children.
///
/// A [ContainerSpan] can have a border, background color, height, width, padding, and
margin
/// and can represent either an INLINE or BLOCK-level element.
class ContainerSpan extends StatelessWidget {
  final AnchorKey? key;
  final Widget? child;
  final List<InlineSpan>? children;
  final Style style;
  final RenderContext newContext;
  final bool shrinkWrap;

  ContainerSpan({
    this.key,
    this.child,
    this.children,
    required this.style,
    required this.newContext,
    this.shrinkWrap = false,
  }): super(key: key);

  @override
  Widget build(BuildContext _) {
    return Container(
      decoration: BoxDecoration(
        border: style.border,
        color: style.backgroundColor,
      ),
      height: style.height,
      width: style.width,
```

```
      padding: style.padding?.nonNegative,
      margin: style.margin?.nonNegative,
      alignment: shrinkWrap ? null : style.alignment,
      child: child ??
          StyledText(
            textSpan: TextSpan(
              style: newContext.style.generateTextStyle(),
              children: children,
            ),
            style: newContext.style,
            renderContext: newContext,
          ),
    );
  }
}

class StyledText extends StatelessWidget {
  final InlineSpan textSpan;
  final Style style;
  final double textScaleFactor;
  final RenderContext renderContext;
  final AnchorKey? key;
  final bool _selectable;
  final TextSelectionControls? selectionControls;
  final ScrollPhysics? scrollPhysics;

  const StyledText({
    required this.textSpan,
    required this.style,
    this.textScaleFactor = 1.0,
    required this.renderContext,
    this.key,
    this.selectionControls,
    this.scrollPhysics,
  }) : _selectable = false,
       super(key: key);

  const StyledText.selectable({
    required TextSpan textSpan,
    required this.style,
    this.textScaleFactor = 1.0,
    required this.renderContext,
    this.key,
    this.selectionControls,
    this.scrollPhysics,
  }) : textSpan = textSpan,
       _selectable = true,
       super(key: key);
```

```
  @override
  Widget build(BuildContext context) {
    if (_selectable) {
      return SelectableText.rich(
        textSpan as TextSpan,
        style: style.generateTextStyle(),
        textAlign: style.textAlign,
        textDirection: style.direction,
        textScaleFactor: textScaleFactor,
        maxLines: style.maxLines,
        selectionControls: selectionControls,
        scrollPhysics: scrollPhysics,
      );
    }
    return SizedBox(
      width: consumeExpandedBlock(style.display, renderContext),
      child: Text.rich(
        textSpan,
        style: style.generateTextStyle(),
        textAlign: style.textAlign,
        textDirection: style.direction,
        textScaleFactor: textScaleFactor,
        maxLines: style.maxLines,
        overflow: style.textOverflow,
      ),
    );
  }

  double? consumeExpandedBlock(Display? display, RenderContext context) {
    if ((display == Display.BLOCK || display == Display.LIST_ITEM) &&
!renderContext.parser.shrinkWrap) {
      return double.infinity;
    }
    return null;
  }
}

extension IterateLetters on String {
  String nextLetter() {
    String s = this.toLowerCase();
    if (s == "z") {
      return String.fromCharCode(s.codeUnitAt(0) - 25) +
String.fromCharCode(s.codeUnitAt(0) - 25); // AA or aa
    } else {
      var lastChar = s.substring(s.length - 1);
      var sub = s.substring(0, s.length - 1);
      if (lastChar == "z") {
```

```
      // If a string of length > 1 ends in Z/z,
      // increment the string (excluding the last Z/z) recursively,
      // and append A/a (depending on casing) to it
      return sub.nextLetter() + 'a';
    } else {
      // (take till last char) append with (increment last char)
      return sub + String.fromCharCode(lastChar.codeUnitAt(0) + 1);
    }
  }
 }
}
```

Material.dart –

```
// Copyright 2014 The Flutter Authors. All rights reserved.
// Use of this source code is governed by a BSD-style license that can be
// found in the LICENSE file.

/// Flutter widgets implementing Material Design.
///
/// To use, import `package:flutter/material.dart`.
///
/// {@youtube 560 315 https://www.youtube.com/watch?v=DL0Ix1lnC4w}
///
/// See also:
///
///  *
[docs.flutter.dev/ui/widgets/material](https://docs.flutter.dev/ui/widgets/
material)
///    for a catalog of commonly-used Material component widgets.
///  * [m3.material.io](https://m3.material.io/) for the Material 3
specification
///  * [m2.material.io](https://m2.material.io/) for the Material 2
specification
library material;

export 'src/material/about.dart';
export 'src/material/action_buttons.dart';
export 'src/material/action_chip.dart';
export 'src/material/action_icons_theme.dart';
export 'src/material/adaptive_text_selection_toolbar.dart';
export 'src/material/animated_icons.dart';
export 'src/material/app.dart';
export 'src/material/app_bar.dart';
export 'src/material/app_bar_theme.dart';
export 'src/material/arc.dart';
export 'src/material/autocomplete.dart';
export 'src/material/badge.dart';
export 'src/material/badge_theme.dart';
export 'src/material/banner.dart';
export 'src/material/banner_theme.dart';
export 'src/material/bottom_app_bar.dart';
```

```
export 'src/material/bottom_app_bar_theme.dart';
export 'src/material/bottom_navigation_bar.dart';
export 'src/material/bottom_navigation_bar_theme.dart';
export 'src/material/bottom_sheet.dart';
export 'src/material/bottom_sheet_theme.dart';
export 'src/material/button.dart';
export 'src/material/button_bar.dart';
export 'src/material/button_bar_theme.dart';
export 'src/material/button_style.dart';
export 'src/material/button_style_button.dart';
export 'src/material/button_theme.dart';
export 'src/material/calendar_date_picker.dart';
export 'src/material/card.dart';
export 'src/material/card_theme.dart';
export 'src/material/checkbox.dart';
export 'src/material/checkbox_list_tile.dart';
export 'src/material/checkbox_theme.dart';
export 'src/material/chip.dart';
export 'src/material/chip_theme.dart';
export 'src/material/choice_chip.dart';
export 'src/material/circle_avatar.dart';
export 'src/material/color_scheme.dart';
export 'src/material/colors.dart';
export 'src/material/constants.dart';
export 'src/material/curves.dart';
export 'src/material/data_table.dart';
export 'src/material/data_table_source.dart';
export 'src/material/data_table_theme.dart';
export 'src/material/date.dart';
export 'src/material/date_picker.dart';
export 'src/material/date_picker_theme.dart';
export 'src/material/debug.dart';
export 'src/material/desktop_text_selection.dart';
export 'src/material/desktop_text_selection_toolbar.dart';
export 'src/material/desktop_text_selection_toolbar_button.dart';
export 'src/material/dialog.dart';
export 'src/material/dialog_theme.dart';
export 'src/material/divider.dart';
export 'src/material/divider_theme.dart';
export 'src/material/drawer.dart';
export 'src/material/drawer_header.dart';
export 'src/material/drawer_theme.dart';
export 'src/material/dropdown.dart';
export 'src/material/dropdown_menu.dart';
export 'src/material/dropdown_menu_theme.dart';
export 'src/material/elevated_button.dart';
export 'src/material/elevated_button_theme.dart';
export 'src/material/elevation_overlay.dart';
export 'src/material/expand_icon.dart';
export 'src/material/expansion_panel.dart';
export 'src/material/expansion_tile.dart';
export 'src/material/expansion_tile_theme.dart';
export 'src/material/filled_button.dart';
```

```
export 'src/material/filled_button_theme.dart';
export 'src/material/filter_chip.dart';
export 'src/material/flexible_space_bar.dart';
export 'src/material/floating_action_button.dart';
export 'src/material/floating_action_button_location.dart';
export 'src/material/floating_action_button_theme.dart';
export 'src/material/flutter_logo.dart';
export 'src/material/grid_tile.dart';
export 'src/material/grid_tile_bar.dart';
export 'src/material/icon_button.dart';
export 'src/material/icon_button_theme.dart';
export 'src/material/icons.dart';
export 'src/material/ink_decoration.dart';
export 'src/material/ink_highlight.dart';
export 'src/material/ink_ripple.dart';
export 'src/material/ink_sparkle.dart';
export 'src/material/ink_splash.dart';
export 'src/material/ink_well.dart';
export 'src/material/input_border.dart';
export 'src/material/input_chip.dart';
export 'src/material/input_date_picker_form_field.dart';
export 'src/material/input_decorator.dart';
export 'src/material/list_tile.dart';
export 'src/material/list_tile_theme.dart';
export 'src/material/magnifier.dart';
export 'src/material/material.dart';
export 'src/material/material_button.dart';
export 'src/material/material_localizations.dart';
export 'src/material/material_state.dart';
export 'src/material/material_state_mixin.dart';
export 'src/material/menu_anchor.dart';
export 'src/material/menu_bar_theme.dart';
export 'src/material/menu_button_theme.dart';
export 'src/material/menu_style.dart';
export 'src/material/menu_theme.dart';
export 'src/material/mergeable_material.dart';
export 'src/material/motion.dart';
export 'src/material/navigation_bar.dart';
export 'src/material/navigation_bar_theme.dart';
export 'src/material/navigation_drawer.dart';
export 'src/material/navigation_drawer_theme.dart';
export 'src/material/navigation_rail.dart';
export 'src/material/navigation_rail_theme.dart';
export 'src/material/no_splash.dart';
export 'src/material/outlined_button.dart';
export 'src/material/outlined_button_theme.dart';
export 'src/material/page.dart';
export 'src/material/page_transitions_theme.dart';
export 'src/material/paginated_data_table.dart';
export 'src/material/popup_menu.dart';
export 'src/material/popup_menu_theme.dart';
export 'src/material/predictive_back_page_transitions_builder.dart';
export 'src/material/progress_indicator.dart';
```

```
export 'src/material/progress_indicator_theme.dart';
export 'src/material/radio.dart';
export 'src/material/radio_list_tile.dart';
export 'src/material/radio_theme.dart';
export 'src/material/range_slider.dart';
export 'src/material/refresh_indicator.dart';
export 'src/material/reorderable_list.dart';
export 'src/material/scaffold.dart';
export 'src/material/scrollbar.dart';
export 'src/material/scrollbar_theme.dart';
export 'src/material/search.dart';
export 'src/material/search_anchor.dart';
export 'src/material/search_bar_theme.dart';
export 'src/material/search_view_theme.dart';
export 'src/material/segmented_button.dart';
export 'src/material/segmented_button_theme.dart';
export 'src/material/selectable_text.dart';
export 'src/material/selection_area.dart';
export 'src/material/shadows.dart';
export 'src/material/slider.dart';
export 'src/material/slider_theme.dart';
export 'src/material/snack_bar.dart';
export 'src/material/snack_bar_theme.dart';
export 'src/material/spell_check_suggestions_toolbar.dart';
export 'src/material/spell_check_suggestions_toolbar_layout_delegate.dart';
export 'src/material/stepper.dart';
export 'src/material/switch.dart';
export 'src/material/switch_list_tile.dart';
export 'src/material/switch_theme.dart';
export 'src/material/tab_bar_theme.dart';
export 'src/material/tab_controller.dart';
export 'src/material/tab_indicator.dart';
export 'src/material/tabs.dart';
export 'src/material/text_button.dart';
export 'src/material/text_button_theme.dart';
export 'src/material/text_field.dart';
export 'src/material/text_form_field.dart';
export 'src/material/text_selection.dart';
export 'src/material/text_selection_theme.dart';
export 'src/material/text_selection_toolbar.dart';
export 'src/material/text_selection_toolbar_text_button.dart';
export 'src/material/text_theme.dart';
export 'src/material/theme.dart';
export 'src/material/theme_data.dart';
export 'src/material/time.dart';
export 'src/material/time_picker.dart';
export 'src/material/time_picker_theme.dart';
export 'src/material/toggle_buttons.dart';
export 'src/material/toggle_buttons_theme.dart';
export 'src/material/tooltip.dart';
export 'src/material/tooltip_theme.dart';
export 'src/material/tooltip_visibility.dart';
export 'src/material/typography.dart';
```

```
export 'src/material/user_accounts_drawer_header.dart';
export 'widgets.dart';
```

HomecategoryWidget.dart –

```
import 'package:carousel_slider/carousel_slider.dart';
import '../models/CategoryModel.dart';
import '../models/PlaceModel.dart';
import '../utils/Extensions/Commons.dart';
import '../utils/Extensions/Widget_extensions.dart';
import '../utils/Extensions/context_extensions.dart';
import '../utils/Extensions/int_extensions.dart';
import '../utils/Extensions/string_extensions.dart';
import 'package:flutter/material.dart';
import '../main.dart';
import '../models/DashboardResponse.dart';
import '../screens/CategoryScreen.dart';
import '../screens/PlaceDetailScreen.dart';
import '../screens/ViewAllScreen.dart';
import 'package:carousel_slider/carousel_controller.dart';
import '../utils/AppColor.dart';
import '../utils/Common.dart';
import '../utils/Extensions/Colors.dart';
import '../utils/Extensions/Constants.dart';
import '../utils/Extensions/decorations.dart';
import '../utils/Extensions/text_styles.dart';

class HomeCategoryWidget extends StatefulWidget {
 final List<CategoryPlaceModel> categoryPlaceModel;

 HomeCategoryWidget(this.categoryPlaceModel);

 @override
 HomeCategoryWidgetState createState() => HomeCategoryWidgetState();
}

class HomeCategoryWidgetState extends State<HomeCategoryWidget> {
 CarouselController controller = CarouselController();
 String? selectedCatId = "";
 int categoryIndex = 1;
 int placeIndex = 1;

 @override
 void initState() {
   super.initState();
   init();
 }

 void init() async {
   //
 }

 @override
```

```
 void setState(fn) {
   if (mounted) super.setState(fn);
 }

 @override
 Widget build(BuildContext context) {
   return widget.categoryPlaceModel.isNotEmpty
       ? Column(
           children: [
             headingWidget(language.category, () {
               CategoryScreen().launch(context);
             }),
             16.height,
             CarouselSlider.builder(
               carouselController: controller,
               itemCount: widget.categoryPlaceModel.length,
               itemBuilder: (BuildContext context, int itemIndex, int
pageViewIndex) {
                 CategoryModel category =
widget.categoryPlaceModel[itemIndex].category!;
                 return GestureDetector(
                   onTap: () {
                     ViewAllScreen(name: category.name.validate(), catId:
category.id).launch(context);
                   },
                   child: Column(
                     children: [
                       cachedImage(category.image.validate(), width:
context.width() / 3, fit:
BoxFit.cover).cornerRadiusWithClipRRect(defaultRadius).expand(),
                       10.height,
                       Text(
                         category.name.toString(),
                         style: itemIndex == categoryIndex ?
boldTextStyle(color: primaryColor) : primaryTextStyle(),
                         textAlign: TextAlign.center,
                         maxLines: 1,
                         overflow: TextOverflow.ellipsis,
                       ),
                     ],
                   ).paddingOnly(left: 6, right: 6),
                 );
               },
               options: CarouselOptions(
                 enlargeCenterPage: true,
                 enableInfiniteScroll: false,
                 onPageChanged: (i, v) {
                   selectedCatId =
widget.categoryPlaceModel[categoryIndex].category!.id;
                   categoryIndex = i;
                   placeIndex = 1;
                   setState(() {});
                 },
```

```
                initialPage: categoryIndex,
                height: context.width() / 3,
                viewportFraction: 0.33,
              ),
            ),
            20.height,
            (widget.categoryPlaceModel[categoryIndex].places ??
[]).isNotEmpty
                ? CarouselSlider.builder(
                    carouselController: controller,
                    itemCount:
widget.categoryPlaceModel[categoryIndex].places!.length,
                    itemBuilder: (BuildContext context, int itemIndex, int
pageViewIndex) {
                      PlaceModel place =
widget.categoryPlaceModel[categoryIndex].places![itemIndex];
                      return Stack(
                        alignment: Alignment.center,
                        clipBehavior: Clip.none,
                        children: [
                          cachedImage(place.image.toString(), height:
context.height(), width: context.width(), fit:
BoxFit.cover).cornerRadiusWithClipRRect(defaultRadius).paddingOnly(bottom:
24),
                          Container(
                            margin: EdgeInsets.only(bottom: 24),
                            height: context.height(),
                            width: context.width(),
                            decoration:
boxDecorationWithRoundedCornersWidget(
                              borderRadius: radius(defaultRadius),
                              border: Border.all(color: appStore.isDarkMode
? Colors.white.withOpacity(0.1) : Colors.transparent),
                              gradient: LinearGradient(
                                begin: Alignment.topCenter,
                                end: Alignment.bottomCenter,
                                colors: [
                                  Colors.black.withOpacity(0.1),
                                  Colors.black.withOpacity(0.1),
                                  Colors.black.withOpacity(0.3),
                                  Colors.black.withOpacity(0.9),
                                ],
                              ),
                            ),
                          ),
                          Positioned(
                            top: 16,
                            right: 16,
                            child: GestureDetector(
                              child: favouriteItemWidget(placeId:
place.id.validate()),
                              onTap: () {
```

```
appStore.setSelectedPlaceId(place.id.validate());
                                appStore.addRemoveFavouriteData(context,
place.id.validate());
                                setState(() {});
                              },
                            ),
                          ),
                          Positioned(
                            left: 16,
                            right: 16,
                            bottom: 0,
                            child: Column(
                              children: [
                                Text(place.name.toString().toUpperCase(),
style: boldTextStyle(color: whiteColor, size: 18), textAlign:
TextAlign.center, maxLines: 2),
                                8.height,
                                Text(

parseHtmlStringWidget(place.description.toString()),
                                  style: secondaryTextStyle(color:
whiteColor),

                                  maxLines: 3,
                                  overflow: TextOverflow.ellipsis,
                                  textAlign: TextAlign.center,
                                ).visible(place.description != null),
                                16.height,
                                Container(
                                  padding: EdgeInsets.symmetric(horizontal:
20, vertical: 8),
                                  decoration:
boxDecorationWithRoundedCornersWidget(backgroundColor: context.cardColor,
boxShadow: [BoxShadow(blurRadius: 1)], borderRadius: radius(defaultRadius)),
                                  child: Text(language.explore, style:
boldTextStyle()),
                                ),
                                8.height,
                              ],
                            ),
                          ),
                        ],
                      ).onTap(() {
                        appStore.setSelectedPlaceId(place.id.validate());
                        PlaceDetailScreen(placeData: place).launch(context);
                      });
                    },
                    options: CarouselOptions(
                      enlargeCenterPage: true,
                      enableInfiniteScroll: false,
                      onPageChanged: (i, v) {
                        placeIndex = i;
                        setState(() {});
```

```
                        },
                        initialPage: placeIndex,
                        viewportFraction: 0.7,
                        aspectRatio: 1,
                      ),
                    )
                  : emptyWidget(),
              20.height,
            ],
          )
        : emptyWidget();
  }
}
```

Carousel_slider.dart –

```
library carousel_slider;

import 'dart:async';

import 'package:carousel_slider/carousel_state.dart';
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';

import 'carousel_controller.dart';
import 'carousel_options.dart';
import 'utils.dart';

export 'carousel_controller.dart';
export 'carousel_options.dart';

typedef Widget ExtendedIndexedWidgetBuilder(
    BuildContext context, int index, int realIndex);

class CarouselSlider extends StatefulWidget {
  /// [CarouselOptions] to create a [CarouselState] with
  final CarouselOptions options;

  final bool? disableGesture;

  /// The widgets to be shown in the carousel of default constructor
  final List<Widget>? items;

  /// The widget item builder that will be used to build item on demand
  /// The third argument is the PageView's real index, can be used to
cooperate
  /// with Hero.
  final ExtendedIndexedWidgetBuilder? itemBuilder;

  /// A [MapController], used to control the map.
  final CarouselControllerImpl _carouselController;

  final int? itemCount;
```

```
 CarouselSlider(
     {required this.items,
     required this.options,
     this.disableGesture,
     CarouselController? carouselController,
     Key? key})
     : itemBuilder = null,
       itemCount = items != null ? items.length : 0,
       _carouselController = carouselController != null
           ? carouselController as CarouselControllerImpl
           : CarouselController() as CarouselControllerImpl,
       super(key: key);

 /// The on demand item builder constructor
 CarouselSlider.builder(
     {required this.itemCount,
     required this.itemBuilder,
     required this.options,
     this.disableGesture,
     CarouselController? carouselController,
     Key? key})
     : items = null,
       _carouselController = carouselController != null
           ? carouselController as CarouselControllerImpl
           : CarouselController() as CarouselControllerImpl,
       super(key: key);

 @override
 CarouselSliderState createState() =>
CarouselSliderState(_carouselController);
}

class CarouselSliderState extends State<CarouselSlider>
   with TickerProviderStateMixin {
 final CarouselControllerImpl carouselController;
 Timer? timer;

 CarouselOptions get options => widget.options;

 CarouselState? carouselState;

 PageController? pageController;

 /// mode is related to why the page is being changed
 CarouselPageChangedReason mode = CarouselPageChangedReason.controller;

 CarouselSliderState(this.carouselController);

 void changeMode(CarouselPageChangedReason _mode) {
   mode = _mode;
 }
```

```dart
@override
void didUpdateWidget(CarouselSlider oldWidget) {
  carouselState!.options = options;
  carouselState!.itemCount = widget.itemCount;

  // pageController needs to be re-initialized to respond to state changes
  pageController = PageController(
    viewportFraction: options.viewportFraction,
    initialPage: carouselState!.realPage,
  );
  carouselState!.pageController = pageController;

  // handle autoplay when state changes
  handleAutoPlay();

  super.didUpdateWidget(oldWidget);
}

@override
void initState() {
  super.initState();
  carouselState =
      CarouselState(this.options, clearTimer, resumeTimer, this.changeMode);

  carouselState!.itemCount = widget.itemCount;
  carouselController.state = carouselState;
  carouselState!.initialPage = widget.options.initialPage;
  carouselState!.realPage = options.enableInfiniteScroll
      ? carouselState!.realPage + carouselState!.initialPage
      : carouselState!.initialPage;
  handleAutoPlay();

  pageController = PageController(
    viewportFraction: options.viewportFraction,
    initialPage: carouselState!.realPage,
  );

  carouselState!.pageController = pageController;
}

Timer? getTimer() {
  return widget.options.autoPlay
      ? Timer.periodic(widget.options.autoPlayInterval, (_) {
          if (!mounted) {
            clearTimer();
            return;
          }

          final route = ModalRoute.of(context);
          if (route?.isCurrent == false) {
            return;
          }
```

```
          CarouselPageChangedReason previousReason = mode;
          changeMode(CarouselPageChangedReason.timed);
          int nextPage = carouselState!.pageController!.page!.round() + 1;
          int itemCount = widget.itemCount ?? widget.items!.length;

          if (nextPage >= itemCount &&
              widget.options.enableInfiniteScroll == false) {
            if (widget.options.pauseAutoPlayInFiniteScroll) {
              clearTimer();
              return;
            }
            nextPage = 0;
          }

          carouselState!.pageController!
              .animateToPage(nextPage,
                  duration: widget.options.autoPlayAnimationDuration,
                  curve: widget.options.autoPlayCurve)
              .then((_) => changeMode(previousReason));
        })
      : null;
}

void clearTimer() {
  if (timer != null) {
    timer?.cancel();
    timer = null;
  }
}

void resumeTimer() {
  if (timer == null) {
    timer = getTimer();
  }
}

void handleAutoPlay() {
  bool autoPlayEnabled = widget.options.autoPlay;

  if (autoPlayEnabled && timer != null) return;

  clearTimer();
  if (autoPlayEnabled) {
    resumeTimer();
  }
}

Widget getGestureWrapper(Widget child) {
  Widget wrapper;
  if (widget.options.height != null) {
    wrapper = Container(height: widget.options.height, child: child);
  } else {
    wrapper =
```

```
        AspectRatio(aspectRatio: widget.options.aspectRatio, child: child);
  }

  if (true == widget.disableGesture) {
    return NotificationListener(
      onNotification: (Notification notification) {
        if (widget.options.onScrolled != null &&
            notification is ScrollUpdateNotification) {
          widget.options.onScrolled!(carouselState!.pageController!.page);
        }
        return false;
      },
      child: wrapper,
    );
  }

  return RawGestureDetector(
    behavior: HitTestBehavior.opaque,
    gestures: {
      _MultipleGestureRecognizer:
          GestureRecognizerFactoryWithHandlers<_MultipleGestureRecognizer>(
              () => _MultipleGestureRecognizer(),
              (_MultipleGestureRecognizer instance) {
        instance.onStart = (_) {
          onStart();
        };
        instance.onDown = (_) {
          onPanDown();
        };
        instance.onEnd = (_) {
          onPanUp();
        };
        instance.onCancel = () {
          onPanUp();
        };
      }),
    },
    child: NotificationListener(
      onNotification: (Notification notification) {
        if (widget.options.onScrolled != null &&
            notification is ScrollUpdateNotification) {
          widget.options.onScrolled!(carouselState!.pageController!.page);
        }
        return false;
      },
      child: wrapper,
    ),
  );
}

Widget getCenterWrapper(Widget child) {
  if (widget.options.disableCenter) {
    return Container(
```

```
      child: child,
    );
  }
  return Center(child: child);
}

Widget getEnlargeWrapper(Widget? child,
    {double? width,
    double? height,
    double? scale,
    required double itemOffset}) {
  if (widget.options.enlargeStrategy == CenterPageEnlargeStrategy.height) {
    return SizedBox(child: child, width: width, height: height);
  }
  if (widget.options.enlargeStrategy == CenterPageEnlargeStrategy.zoom) {
    late Alignment alignment;
    final bool horizontal = options.scrollDirection == Axis.horizontal;
    if (itemOffset > 0) {
      alignment = horizontal ? Alignment.centerRight :
Alignment.bottomCenter;
    } else {
      alignment = horizontal ? Alignment.centerLeft : Alignment.topCenter;
    }
    return Transform.scale(child: child, scale: scale!, alignment:
alignment);
  }
  return Transform.scale(
      scale: scale!,
      child: Container(child: child, width: width, height: height));
}

void onStart() {
  changeMode(CarouselPageChangedReason.manual);
}

void onPanDown() {
  if (widget.options.pauseAutoPlayOnTouch) {
    clearTimer();
  }

  changeMode(CarouselPageChangedReason.manual);
}

void onPanUp() {
  if (widget.options.pauseAutoPlayOnTouch) {
    resumeTimer();
  }
}

@override
void dispose() {
  super.dispose();
  clearTimer();
```

```
 }

 @override
 Widget build(BuildContext context) {
   return getGestureWrapper(PageView.builder(
     padEnds: widget.options.padEnds,
     scrollBehavior: ScrollConfiguration.of(context).copyWith(
       scrollbars: false,
       overscroll: false,
       dragDevices: {
         PointerDeviceKind.touch,
         PointerDeviceKind.mouse,
       },
     ),
     clipBehavior: widget.options.clipBehavior,
     physics: widget.options.scrollPhysics,
     scrollDirection: widget.options.scrollDirection,
     pageSnapping: widget.options.pageSnapping,
     controller: carouselState!.pageController,
     reverse: widget.options.reverse,
     itemCount: widget.options.enableInfiniteScroll ? null :
widget.itemCount,
     key: widget.options.pageViewKey,
     onPageChanged: (int index) {
       int currentPage = getRealIndex(index + carouselState!.initialPage,
           carouselState!.realPage, widget.itemCount);
       if (widget.options.onPageChanged != null) {
         widget.options.onPageChanged!(currentPage, mode);
       }
     },
     itemBuilder: (BuildContext context, int idx) {
       final int index = getRealIndex(idx + carouselState!.initialPage,
           carouselState!.realPage, widget.itemCount);

       return AnimatedBuilder(
         animation: carouselState!.pageController!,
         child: (widget.items != null)
             ? (widget.items!.length > 0 ? widget.items![index] :
Container())
             : widget.itemBuilder!(context, index, idx),
         builder: (BuildContext context, child) {
           double distortionValue = 1.0;
           // if `enlargeCenterPage` is true, we must calculate the carousel
item's height
           // to display the visual effect
           double itemOffset = 0;
           if (widget.options.enlargeCenterPage != null &&
               widget.options.enlargeCenterPage == true) {
             // pageController.page can only be accessed after the first
build,
             // so in the first build we calculate the itemoffset manually
             var position = carouselState?.pageController?.position;
             if (position != null &&
```

```
                position.hasPixels &&
                position.hasContentDimensions) {
              var _page = carouselState?.pageController?.page;
              if (_page != null) {
                itemOffset = _page - idx;
              }
            } else {
              BuildContext storageContext = carouselState!
                  .pageController!.position.context.storageContext;
              final double? previousSavedPosition =
                  PageStorage.of(storageContext)?.readState(storageContext)
                      as double?;
              if (previousSavedPosition != null) {
                itemOffset = previousSavedPosition - idx.toDouble();
              } else {
                itemOffset =
                    carouselState!.realPage.toDouble() - idx.toDouble();
              }
            }

            final double enlargeFactor =
                options.enlargeFactor.clamp(0.0, 1.0);
            final num distortionRatio =
                (1 - (itemOffset.abs() * enlargeFactor)).clamp(0.0, 1.0);
            distortionValue =
                Curves.easeOut.transform(distortionRatio as double);
          }

          final double height = widget.options.height ??
              MediaQuery.of(context).size.width *
                  (1 / widget.options.aspectRatio);

          if (widget.options.scrollDirection == Axis.horizontal) {
            return getCenterWrapper(getEnlargeWrapper(child,
                height: distortionValue * height,
                scale: distortionValue,
                itemOffset: itemOffset));
          } else {
            return getCenterWrapper(getEnlargeWrapper(child,
                width: distortionValue * MediaQuery.of(context).size.width,
                scale: distortionValue,
                itemOffset: itemOffset));
          }
        },
      );
    },
  ));
 }
}

class _MultipleGestureRecognizer extends PanGestureRecognizer {}
```